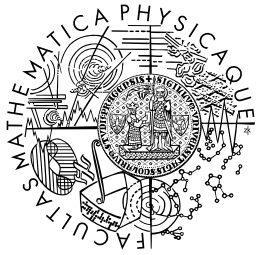


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Vojtech Gáborík

AI Bots Battle Royale

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Jiří Švancara, Ph.D

Study programme: Computer Science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank and dedicate this thesis to my supervisor, the faculty, my family and friends.

Title: AI Bots Battle Royale

Author: Vojtech Gáborík

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Švancara, Ph.D, Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis presents a 2D top-down shooter game featuring AI-controlled teams that compete in a structured environment. Each team is driven by a distinct artificial intelligence strategy: rule-based logic, planning algorithms, neural networks, or evolutionary approaches. Bots navigate a maze-like arena, detect enemies and execute coordinated actions based on their AI design. The system supports customizable simulations, allowing performance comparisons between AI types. The results highlight the strengths and weaknesses of each AI method.

Keywords: Top down shooter, AI agents, Planning agent, Evolution agent, Neural agent, Automata agent

Název práce: AI Bots Battle Royale

Autor: Vojtech Gáborík

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jiří Švancara, Ph.D, Katedra teoretické informatiky a matematické logiky

Abstrakt: Tato práce představuje 2D střílečku s pohledem shora, v níž týmy ovládané umělou inteligencí soutěží ve strukturovaném prostředí. Každý tým je řízen odlišnou strategií umělé inteligence: logikou založenou na pravidlech, plánovacími algoritmy, neuronovými sítěmi nebo evolučními přístupy. Boti se pohybují v aréně podobné bludiště, detekují nepřátele a provádějí koordinované akce na základě svého návrhu umělé inteligence. Systém podporuje přizpůsobitelné simulace, což umožňuje porovnání výkonu mezi typy umělé inteligence. Výsledky zdůrazňují silné a slabé stránky každé metody umělé inteligence.

Klíčová slova: střílečka s pohledem shora, AI agenti, Plánovací agent, Evoluční agent, Neurální agent, Automat agent

Contents

Introduction **8**

1 Game Design **9**
1.1 Core . . . 9
1.2 Round-Based Structure . . . 9
1.3 Maze-Based Movements . . . 9
1.4 Team-Based Combat . . . 9
1.5 Health . . . 10
1.6 Limited information . . . 10
1.7 Player Incentives . . . 10

2 Code Architecture **11**
2.1 Game Flow Layer . . . 11
2.2 Team and Bot Layer . . . 11
2.3 Player Layer . . . 11
2.4 Environment Layer . . . 12
2.5 AI Layer . . . 12
2.6 Simulation . . . 12
2.7 Summary . . . 13

3 Bots and Teams **14**
3.1 Bot Structure and State . . . 14
3.2 Navigation and Pathfinding . . . 14
3.3 Team Communication . . . 15
3.4 Bot Lifecycle and Cleanup . . . 15
3.5 Field of View . . . 15
3.6 Player Component . . . 16
3.7 Bot actions . . . 16

4 Environment **18**
4.1 Grid Representation . . . 18
4.2 Maze Generation . . . 18
4.2.1 Build inter walls with Kruskal . . . 18
4.2.2 Carve regular loops . . . 19
4.2.3 Outer layers . . . 19
4.2.4 Chceck unvisited cells . . . 19
4.2.5 DFS-Based Alternative . . . 19

5 Bot AI **21**
5.1 RandomAI . . . 21
5.2 AutomataAI . . . 21
5.2.1 State Enumeration . . . 21
5.2.2 Randomized Exploration . . . 21
5.2.3 Event-Driven Transitions . . . 22
5.3 PlanningAI . . . 22
5.3.1 Internal State . . . 22

5.3.2 Goals . . . 22
5.3.3 Sensors . . . 23
5.3.4 Actions . . . 23
5.3.5 Executors . . . 23
5.3.6 Planning and Execution Loop . . . 24
5.4 GeneticPlanningAI . . . 24
5.4.1 Parameter Container . . . 24
5.4.2 Wrapping PlanningAI . . . 24
5.5 Evolution Algorithm . . . 24
5.5.1 Setup and Parameters . . . 24
5.5.2 Chromosome Encoding . . . 25
5.5.3 Initial Population . . . 25
5.5.4 Applying Parameters . . . 25
5.5.5 Scoring & Fitness . . . 25
5.5.6 Convergence . . . 26
5.5.7 Selection . . . 26
5.5.8 Crossover . . . 27
5.5.9 Next Generation . . . 27
5.6 NeuralAI . . . 27
5.6.1 Input Layer and State Encoding . . . 27
5.6.2 Hidden Layer . . . 28
5.6.3 Forward Propagation and Decision Making . . . 28
5.7 NeuralAI training . . . 28
5.7.1 Reinforcement Learning Setup in the Simulation . . . 28
5.7.2 `TrainNeuralNetwork` and `TrainOnReward` functions . . . 29
5.7.3 Training Cycle and Weight Updates Frequency . . . 31

6 Game flow **32**
6.1 Game Manager . . . 32
6.1.1 Match Flow and Game State . . . 32
6.1.2 Working with Maze and Teams . . . 32
6.1.3 Deciding When to End the Match . . . 32
6.1.4 Preparing Data for Pathfinding . . . 32
6.1.5 Ending the Game and Cleanup . . . 33
6.2 Simulation . . . 33

7 Experiments **34**
7.1 Experimental Setup . . . 34
7.2 Metrics . . . 34
7.3 Results Summary . . . 34
7.4 1v1 results . . . 36

8 Optimizations **38**
8.1 Parallel Execution . . . 38
8.2 Conditional AI Updates . . . 38

Conclusion **39**

Bibliography **40**

A User Documentation **41**
A.1 Simulation Startup 41
A.2 Game Startup 41
A.3 Menu Structure 41
A.3.1 Team Configuration 41
A.3.2 Maze Configuration 42
A.3.3 AI Toggle 42
A.3.4 Play Button 42
A.4 Maze Editing 42
A.4.1 Perfect Maze Toggle 42
A.4.2 Adding and Removing Walls 43
A.4.3 Moving Team Spawn Points 43
A.4.4 Finalizing the Maze 43
A.5 Gameplay Mechanics 43
A.5.1 Movement 43
A.5.2 Aiming 43
A.5.3 Shooting 44

Introduction

Background and Motivation

During my studies, I have taken several computer games courses. One of the projects I worked on was creating a simple game in the style of a top-down 2D shooter. This game involved navigating a maze, aiming, and dealing with pac-man-styled opponents. Although the game was not overly complex, I quickly realized it had much more potential, particularly when it came to the design of the opponents. This presented an opportunity to look into the design of bots. Given the maze navigation and opponent interaction, I saw that there was the possibility of exploring bots capable of positioning, simple strategy, and decision-making within the game's mechanics and game design. Therefore, I began a journey to dive deeper into artificial intelligence.

AI has gained significant popularity, making its way into various fields. It has found its way even into gaming. The game I created was in need of AI-controlled bots. When I began exploring how to implement artificial intelligence, I discovered an overwhelming number of potential approaches to solve this problem. My interest in experimenting with multiple techniques led me to choose a few popular approaches that seemed suitable for my game. The challenge then became how to tailor these different approaches to fit my game and compare them. The idea was to equip different teams of bots with distinct types of AI and pit them against each other to evaluate their performance.

Objective

This thesis will explore how I approached these challenges. I will discuss the details of the game design and gameplay mechanics. The exploration will include a look into the control subsystems of a bot, which manage various functionalities such as sight, aiming, maze navigation, and movement. Furthermore, I will describe the different approaches to artificial intelligence that I employed and provide a comparative analysis. Through this examination, I aim to learn about AI in games and share my experiences when it comes to AI implementation in gaming and the potential benefits and limitations of various techniques.

1 Game Design

The game developed for this thesis is a 2D top-down shooter in which teams of bots, or optionally a human player, engage in combat within a procedurally generated maze. The design emphasises tactical movement, sight-based shooting, and decision-making with partial observability. It is built with experimentation in mind: matches are short, randomised, and easily repeatable. This makes it ideal for comparing AI techniques. Game is written in C# and developed in Unity version 2020.3.48f1.[1]

1.1 Core

The main goal of each match is simple: survive. Bots (and optionally the player) are assigned to teams, and each team must eliminate all members of the opposing teams to win. The game does not use score-based systems or objectives like capture points—it focuses on pure combat and survival. This makes match outcomes easy to measure (who survived, who didn't).

1.2 Round-Based Structure

Matches are structured into rounds, especially in simulation mode. Each round has a fresh maze, randomised spawn points for teams, and resets all AI and power-up states. This ensures that no single match can dominate the results. Instead, performance is measured over many rounds with varied conditions. The benefit of this system is statistical robustness.

1.3 Maze-Based Movements

The world is not an open arena. Instead, it's a square grid-based maze with walls, corridors, and occasional dead ends. The maze is generated procedurally at the start of each round. This means every match takes place in a different layout, with new chokepoints, flanking routes and spawn points . This choice supports the AI-driven nature of the game. Pathfinding, line-ofsight, and positioning become important. The bots, as well as the user, must constantly adapt their movement to navigate new spaces. The player, too, can't memorise the map as it changes every time. The maze also adds a sense of tension. You never know what's around the next corner, and line-of-sight is frequently broken. Bots can't shoot across the whole map—they have to move, peek, and commit to attacks.

1.4 Team-Based Combat

Each match supports up to six teams, with customizable sizes. A match can be 3v3, 2v2v2, or even a 1v1v1v1v1v1 free-for-all. Each team works separately and their capabilities, like flanking and communication, are dependent on the

type of AI the user chooses for them. Friendly fire is always disabled for all teams and also for the user.

1.5 Health

Each bot has three lives, therefore can be shot three times before death. Bots also have invulnerability to damage right after they get hit. It seems too chaotic to have three projectiles hit a bot or a user at once, and suddenly the game is over so it seemed beneficial to add invulnerability that lasts one-tenth of a second.

1.6 Limited information

Unlike most top-down shooters, bots do not have full knowledge of the map or enemies. Their sight is limited to a cone in front of them, blocked by walls. This creates a fog-of-war effect, even though the camera shows the entire map. The fog-of-war effect only applies to the bots, however user's camera has quite a limited view when playing as one of the bots.

1.7 Player Incentives

- Power-ups are spawned periodically during each match. These are placed randomly in valid walkable cells and give temporary bonuses that last only during the given round.
- Power-ups serve multiple roles: They force bots to leave their hiding spots, change direction or delay helping a teammate.
- From a design standpoint, power-ups prevent bots from becoming too passive. They also create hot spots—areas where multiple bots converge, naturally increasing combat frequency.
- Possible bonuses include added health, more fire-rate and added speed.

2 Code Architecture

The project is structured into distinct, purpose-built components to maximize modularity, ease of testing, and extensibility. Unity's component system is leveraged to keep responsibilities separated. The system is divided into several key layers: Game Flow Layer, AI Layer, Team and Bot Layer, Player Layer, Environment Layer and Simulation Layer.

2.1 Game Flow Layer

This layer handles the main control flow of the game. It decides when a match starts, ends, and how the teams and bots are created. It also watches over time limits, checks win conditions, and prepares the game environment.

- Main scripts: `GameManager.cs`, `TeamManager.cs`
- How it connects: Talks to the Environment Layer to place teams into the maze.

  Creates team `GameObjects` and sets up their bots, which are part of the Team and Bot Layer.

  Every frame, it updates the bots indirectly by calling `UpdatePathfinding()` and `UpdateAI()` on each team.

  In simulation mode, it works closely with the Simulation Layer to run automatic matches.

2.2 Team and Bot Layer

- Purpose: This layer manages teams of bots and tracks what each bot knows. It lets bots share information (like when they see an enemy), handles deaths, and connects bots to their AI and movement systems.
- Main scripts: `TeamManager.cs`, `Bot.cs`, `BotState (inside Bot.cs)`, `Pathfinder.cs`
- How it connects: `GameManager.cs` uses this layer to create bots and place them into the game. Each `Bot.cs` object talks to the Player Layer to move or shoot, and to the AI Layer to make decisions. The team manager uses the `Pathfinder` to calculate movement paths for all bots in the team.

2.3 Player Layer

- Purpose: This layer controls how each bot or player moves, rotates, and shoots. It reads movement commands either from human input or from the AI and turns them into physics actions. The field of view of works like a sensor.

- Main scripts: `Player.cs` `FieldOfView.cs`
- How it connects: `Bot.cs` utilises Player through `Pathfinder` When aiming or shooting, the player uses data from the `BotState`, like enemy position.

2.4 Environment Layer

- Purpose: This layer builds the game world — a maze made from a grid of cells. It handles generating walls and passages, editing the maze, and converting between world coordinates and grid positions. It also stores data that bots use for movement.
- Main scripts: `MazeGenerator.cs`
- How it connects: `MazeGenerator` works mostly independently as it creates the maze as it is instantiated. `Pathfinder.cs` is part of the Team and Bot Layer, but it depends on maze data to work.

2.5 AI Layer

- Purpose: This layer holds all the logic for bot decision-making. Each bot is assigned an AI class that tells it what to do based on what it sees and what goals it has.
- Main scripts: `PlanningAI.cs`, `NeuralAI.cs`, `RandomAI.cs`,

  `GeneticPlanningAI.cs`, `AutomataAI.cs`, `BaseAI` (abstract base for all AI types)
- How it connects: Each bot from the Team and Bot Layer has a BaseAI instance assigned. AI reads the bot's state (like whether it sees an enemy) and uses the bot's actions.

2.6 Simulation

- Purpose: This layer runs the game automatically for many rounds. It is used to train neural networks, evolve AI behaviour, or test different strategies in bulk without human input.
- Main scripts: `Simulation.cs`, `EvolutionAlgorithm.cs`
- How it connects: Calls `GameManager.StartGame()` to begin a round, then waits for it to end. If evolution is enabled, `EvolutionAlgorithm.cs` applies genetic changes and selects better teams each generation. When using neural training, it manages experience and reward collection across matches.

2.7 Summary

The code architecture of this project supports rapid iteration, high-performance simulation, and clean AI testing. By clearly dividing logic between `GameManager`, `MazeGenerator`, `TeamManager`, `Bot` components, and AI modules, each piece can evolve independently. This structure also makes it easier to expand the code with elements such as new AI types or new generating functions for the maze. In short, the system is built like a toolbox.

3 Bots and Teams

3.1 Bot Structure and State

Each bot in the game is composed of multiple modular components that together define its physical attributes and collisions for the Unity engine, sensory capabilities, internal state, and behavioural logic. These components are organized under a single GameObject in Unity and communicate through well-defined interfaces. The primary class responsible for managing a bot is Bot.cs. This script holds key metadata such as the bot's ID, team affiliation, and reference links to other subsystems. The bot maintains an internal state through the BotState object, which tracks a wide range of variables needed for decision-making, including:

- Boolean flags indicating sensory events (e.g., `FoundEnemy` , `TeammateDied` , `RetrievedPowerUp`)
- The bot's own position and rotation (`MyPosition` , `LookingAt`)
- Enemy and power-up locations (`EnemyPosition`, `PowerUpPosition`)
- Navigation target (`Destination`) and a flag for triggering pathfinding (`NeedPathfinding`)

Bots are controlled indirectly via an AI module assigned at runtime, which issues high-level decisions such as selecting destinations or initiating defensive manoeuvres. The AI does not manipulate physics or visuals directly; it delegates those tasks to the Player.cs component. The Player component handles all movement and aiming through Unity's physics system and maintains the active navigation. Visual perception is managed by `FieldOfView.cs`, which casts a cone to detect enemies and power-ups. If a visible object is detected, the bot's internal state is updated accordingly, and the event is reported to the team manager. Team-level coordination is handled by the `TeamManager` component. It propagates enemy sightings and status changes (e.g., bot death) across the team by invoking dedicated callbacks like `TeammateFoundAnEnemy()` or `TeammateDied()` on each bot. This centralised communication design allows bots to react to shared knowledge without direct polling of teammates. Overall, this structure separates sensing, decision-making, movement, and team logic into specialised components, which improves modularity and enables flexible AI swapping and simulation control.

3.2 Navigation and Pathfinding

Bots navigate through the maze by first choosing a destination cell. This could be a random, unexplored area, the last known location of an enemy, or the position of a power-up. Once a target is chosen, the bot sets its `NeedPathfinding` flag to true. Each team has its own Pathfinder component, which checks this flag for every bot. Instead of calculating paths one by one, all requests are collected into a single batch. Then the Pathfinder runs the A* algorithm [2] to compute

the shortest paths for all bots that requested them. This happens in parallel, allowing multiple paths to be solved at the same time. The resulting paths are saved as a stack of `Vector2` waypoints, which the bot follows step-by-step using physics-based movement.

This design avoids unnecessary performance costs by making sure pathfinding is only triggered when needed, rather than on every frame. Bots that don't move or have already reached their destination don't waste computation. Since all requests are processed together in a batch, it greatly reduces overhead compared to doing pathfinding individually for each bot. On top of that, because the Pathfinder does not use Unity API functions, its logic can safely run on background threads, letting the .NET thread pool handle the work [3]. This makes the pathfinding system scalable and efficient, especially in large simulations with many bots acting at once.

Each team having its own Pathfinder also has gameplay benefits. Some AI strategies, such as flanking, temporarily change the maze data to block certain paths. If all teams shared the same maze data, this would cause conflicts. By giving each team its own copy of the maze, the system allows bots to test different path scenarios without interfering with others.

3.3 Team Communication

`TeamManager` holds arrays of all bots and their known enemies. When a bot sees an enemy, it doesn't try to handle teamwork by itself. Instead, it calls `Team.FoundEnemy(botID, enemy)`. The `TeamManager` receives this event and notifies the other bots in the team. This allows them to switch into a "help" mode, where they assist the teammate by heading to the same enemy or preparing to defend. Similar callbacks exist for other important events like `TeammateArrivedToEnemy`, which tells teammates that aid has arrived, and `TeammateDied`, which alerts the team when someone is eliminated.

Centralizing these signals in the `TeamManager` makes squad tactics possible without requiring each bot to constantly check what others are doing. Bots only need to respond to the shared events passed through the manager.

3.4 Bot Lifecycle and Cleanup

When a bot dies, its `OnDestroy()` triggers `TeamManager.TeammateDied()`. The manager cleans up the arrays and informs survivors. This keeps team state accurate even as the battle shifts, so lone bots know when they're last standing.

3.5 Field of View

Each bot in the game uses a custom vision system to see nearby objects. This is handled by the `FieldOfView.cs` script. Rather than each bot scanning the whole scene, a custom mesh-based vision cone is used. Many 2D rays are cast in an 80° arc each frame, building a Mesh to visualise sight [4].

Each ray checks for objects up to a set distance. If a ray hits another bot, it calls `FOVFoundEnemy()` on the bot. This updates the bot's state and notifies

the `TeamManager`, so teammates can react. If the ray hits a power-up, the bot calls `FOVFoundPowerUp()` to store its location for later. Only selected layers are checked during raycasting. The function `SetLayerMask()` sets up which layers to include. It ignores bots on the same team and only checks enemies, power-ups, and walls. Its direction is based on where the bot is aiming. For bots controlled by AI, it points toward the current enemy or waypoint. For human players, it follows the mouse.

3.6 Player Component

Player.cs is responsible for handling how bots and human players move, rotate, and shoot. It separates movement and aiming from decision-making, so that bots can be controlled by different types of AI without changing the movement code. For AI-controlled bots, if there's a path to follow, the script pops the next waypoint from the path stack and calculates a direction vector. This direction is normalised and then multiplied by a speed factor d. The bot moves using: `Body.AddForce(direction * d);` This movement is smooth and works with Unity's physics system. If the bot colides with another teammate or even an enemy, it may move a tiny bit in a random direction by adding a force in that direction. This is the case since bots were occasionally getting stuck as they both wanted to search in exactly opposite directions.

This separation, where `Bot` provides logic, `Player` provides motion and aiming and `FieldOfView` provides sight, keeps each part simple and lets us tweak one without breaking the others.

3.7 Bot actions

The Bot class implements the `IBotActions` interface, which encapsulates core tactical behaviors bots can execute in response to strategic decisions made by their AI controller. These functions define how bots navigate, engage, and respond within the game environment:

- `StartSearch()` Selects a random cell from either the upper or lower corridor of the maze and sets it as the destination. This is typically used for exploration or patrolling when no enemies are detected.
- `StartEnemyDestination()` Updates the bot's destination to the last known position of the currently assigned enemy. It's a direct attack command used in offensive strategies.
- `StartPowerUpDestination()` Sets the destination to the location of a discovered power-up, initiating a retrieval action.
- `TryDefensiveSpot()` Scans the environment between the bot and its enemy to find a viable defensive turn (a T-junction or corridor corner). If successful, sets it as the new destination. Returns true if a valid defensive spot is found.
- `TryEnemyFlank()` Uses the pathfinding system to attempt a flanking route to the enemy by temporarily blocking the direct line-of-sight cells. If an alternate path exists, it's set and returned true.

- Stop() Immediately nullifies the bot's destination, halting movement and disengaging from pathfinding.

These actions form the low-level execution layer of the bot and are called by AI systems depending on their logic and current world state.

4 Environment

The game world is a square grid maze, built and managed by `MazeGenerator`. It lets the user edit walls and place team spawns.

4.1 Grid Representation

The maze is stored as a 2D array of cells with its state, game world position and game object that represents the rigid body with the collider and the sprite of a wall. Each Cell records whether it's a wall or path. Coordinates map from world to grid as this:

`int x = (position.x-startCoordX+wallSize/2)/wallSize)`

and

`int y = (int)((-1*(position.y-startCoordY)+wallSize/2)/wallSize)`

where the start coordinates point to the upper leftmost corner of the maze. This makes it easy for bots to pick a target cell and ask the pathfinder for a route

4.2 Maze Generation

Generating the maze actually supports two approaches to carving the interior: a Kruskal-based spanning tree [5] and a recursive DFS. Below is a breakdown of each phase, plus a look at the DFS variant.

4.2.1 Build inter walls with Kruskal

This method, together with carving, creates a maze suitable for flanking as it creates a connected tree into which cycles are carved:

- Fill interior with walls:

  Every coordinate `(x,y)` with `2`≤`x,y`≤`mazeSize-3` is turned into a solid wall:

- Carve out "cell-centres":

  Every coordinate `(x,y)` with `2`≤`x,y`≤`mazeSize-3` is turned into a solid wall

- Fill the interior with walls:

  Logical cells live on a coarse grid of size `cellCount = (mazeSize-3)/2`. Each centre at `(cx,cy)` maps to world-grid coords `(2*cx+2, 2*cy+2)`, which we clear into passages.

- Union-Find setup:

  Treat each of the `cellCount` centres as a disjoint-set element.

- List all edges:

  For each centre, add an edge to its right neighbour at `wall-coord (2*cx+3, 2*cy+2)` and to its down neighbour at `(2*cx+2, 2*cy+3)`.

- Shuffle and Kruskal's loop:

  Randomly shuffle edges and then destroy the wall at `wallPos`, marking it as a path, for each edge connecting two different sets and make a union of the two sets. After this finishes, all centres are connected in a single spanning tree.

4.2.2 Carve regular loops

Once you have a perfect tree, you often want a few extra loops to allow for better chances for flanking the enemy. This method:

- Iterates over even-even logical cells and, when both neighbours are already open on each side of a vertical wall, removes that wall.
- Similarly for odd–odd cells on horizontal walls.

This selectively carves loops to break the tree and create cycles, ensuring occasional alternative routes to enable flanking for the bots.

The rules for filling these tiles are as follows: All cells indexed `(2x, 2y)` are designated as walls. If there is a wall between `(x, y)` and `(x, y + 1)` in the lower-resolution maze, then a wall cell is placed at `(2x + 1, 2(y + 1))`. If there is a wall between `(x, y)` and `(x + 1, y)`, then a wall cell is placed at `(2(x + 1), 2y + 1)`.

4.2.3 Outer layers

The 3 outer layers of the maze are not handled by maze generation but have their uses. The outermost layer serves as a capsule for the entire maze and the bots. The second layer and the third outermost layers serve to create a good spawnpoint for the bots by allowing the bots to spawn at the second outermost layer while still being filled with walls besides the spawnpoints and the third outermost layer serves to open the way into the maze and quick traversal through it as well as making sure that the spawn points are not blocked at the second outermost layer. Before this was implemented, it would happen quite often that teams would spawn next to each other near the outermost wall, and the issue was that there are fewer places to cover or flank to as there is a wall on one side.

4.2.4 Chceck unvisited cells

Finally, any cell left in state Unvisited (i.e. never explicitly turned into a Wall or Path) is marked as Path. All passages are collected into allPathCells for Pathfinding:

4.2.5 DFS-Based Alternative

Generation involves using Depth-First Search (DFS) with random opening of nodes from the queue to create internal walls and paths. This DFS operates on a lower-resolution grid, where walls act as infinitely thin boundaries between path tiles. To upscale this lower-resolution maze to the final version with thick walls,

the algorithm integrates the placement of wall tiles during the DFS process. In this enhanced approach, DFS skips tiles in the grid, designating these skipped tiles as either walls or paths based on specific rules. The rules for filling these tiles are as follows: All cells indexed `(2x, 2y)` are designated as walls. If there is a wall between `(x, y)` and `(x, y + 1)` in the lower-resolution maze, then a wall cell is placed at `(2x + 1, 2(y + 1))`. If there is a wall between `(x, y)` and `(x + 1, y)`, then a wall cell is placed at `(2(x + 1), 2y + 1)`. This alternative creates a maze without the possibility for flanking withing the maze. It is only possible on the perimeter of the environment.

5 Bot AI

The Bot.cs script implements the core behaviour of an individual autonomous agent and encapsulates the capabilities of an agent. The script maintains a state through the `BotState` object, it provides actions that can be utilised by the given AI, automates communication of a team, and also holds the `BaseAI` abstract class. The Decision-making is delegated to an AI module derived from `BaseAI`, allowing support for multiple AI approaches such as planning-based, genetic, or neural control. In conclusion, the entire decision-making process is based on processing the state of the bot by the AI and executing actions accordingly.

5.1 RandomAI

`RandomAI` serves as a naïve baseline, exhibiting no goal-driven behaviours. Its sole "strategy" is to wander the maze randomly and on the occasion that it encounters an enemy, it chooses random actions. As the simplest possible AI, `RandomAI` provides a performance floor against which more sophisticated approaches (e.g., `AutomataAI`, `PlanningAI`) can be compared.

5.2 AutomataAI

`AutomataAI` is a straightforward finite-state machine (FSM) [6] that drives each bot through a set of discrete behaviours based on in-game events and simple sensory flags. Its key characteristics are:

5.2.1 State Enumeration

`AutomataAI` defines six states:

- `Searching` – roaming the maze until a point of interest is found, such as an enemy or a power up
- `GoingToDefensiveSpot` – moving to cover when an enemy is spotted
- `Defending` – holding position until teammates arrive or conditions change
- `GoingToMyEnemy` – pursuing a known enemy
- `GoingToHelpTeamate` – rushing to assist a teammate who found an enemy
- `GoingToPowerUp` – going toward a power-up when it is detected

5.2.2 Randomized Exploration

In the Searching state, bots call `bot.StartSearch()`, This ensures coverage of the maze without any global planning overhead by `AutomataAI`.

5.2.3 Event-Driven Transitions

Each frame `UpdateAI()`, the FSM checks flags in `bot.State` (e.g., `FoundEnemy`, `TeammateFoundEnemy`, `FoundPowerUp`) and whether the bot has arrived at its current destination with `CheckArrivedToDestination()`. Following figure describes the transitions between states:

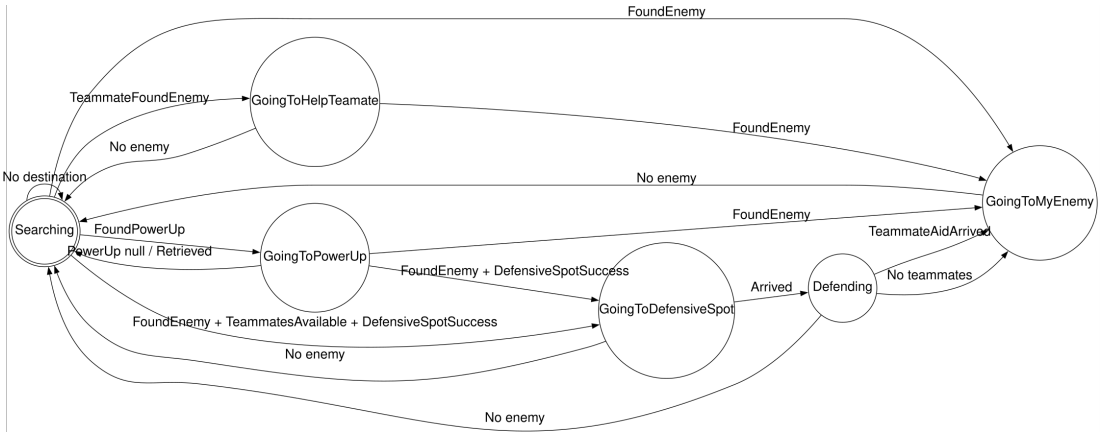


Figure 5.1 Finite State Machine of AutomataAI.

Because it relies solely on local state flags and simple pathfinding calls (no global planning or learning), `AutomataAI` is lightweight and predictable. Ideal as a baseline opponent against which to compare more advanced AI strategies.

5.3 PlanningAI

`PlanningAI` uses Goal-Oriented Action Planning (GOAP) [7] system to pick what the bot should do next. It uses mountain-goap library [8] and it works in these steps:

5.3.1 Internal State

When you create a `PlanningAI`, it makes an Agent with a set of named flags (like `enemyFound`, `rush`, etc.) all starting as false. These flags track what the bot sees or has done so far.

5.3.2 Goals

The agent has five goals, each with a weight you can tweak, and heavier weights make the planner prefer that goal more. The goals are:

- `Stop` (flag "stop")
- `Rush` toward an enemy ("rush")
- `Flank` around an enemy ("flank")
- `Cover` in a defensive spot ("cover")
- `AwaitTeammate` after taking cover ("awaitTeamate")

5.3.3 Sensors

Every frame, sensor callbacks update the agent's flags by reading the bot's state and the state of the team. For example, `EnemySensorHandler` sets `enemyFound` when the bot sees an enemy, and `TeamatesAvailabilitySensorHandler` sets `teamatesAvailable` if a teammate is free to help.

5.3.4 Actions

The planner builds a list of actions. Each action has its name, executor function, preconditions, postconditions and cost. The actions are: find enemy, rush, flank, stop, cover, await teammate, retrieve power up and help teammate. Each action governs the logic that utilizes `IBotActions` provided by the bot if it is appropriate.

5.3.5 Executors

Each action has a function that actually tells the bot what to do. It also governs the success of that function since even if postconditions are met the action is not instant or deterministic meaning it can fail. These are the executors:

- `SearchAction` – uses `bot.StartSearch()` and it is successful if an enemy is found.
- `Rush` – uses `bot.StartEnemyDestination()` to move directly toward the known enemy location.
- `FlankAction` – tries to move around the enemy if the path is short enough. `PlanningAI` checks if a flank is possible via `bot.TryEnemyFlank()` and if the path length is below a given threshold (`flankPathLimit)`, it initiates the flank. The action succeeds when the flank finishes.
- `StopAction` – executes `bot.Stop()` and immediately succeeds. It is used to intentionally stop the bot.
- `CoverAction` – searches for a defensive position near a corridor junction via `bot.TryDefensiveSpot()` and then stops. The action succeeds either after reaching the position.
- `AwaitTeamateAction` – keeps the bot in cover until a teammate arrives to help. It succeeds once help has arrived or teammates are no longer available.
- `RetrievePowerUpAction` – navigates toward a visible power-up using the function. `bot.StartPowerUpDestination()`. If the power up is retrieved or an enemy is detected, the action immediately succeeds and cancels the navigation.
- `HelpTeamateAction` – uses `bot.StartEnemyDestination()` to move to a position where a teammate found an enemy. Once it sees the enemy, the bot sets `Team.TeammateArrivedToEnemy()`, signaling that aid has arrived, and the action succeeds.

5.3.6 Planning and Execution Loop

On each AI update, `PlanningAI` first checks if the bot has reached its current destination, then calls `agent.Step()` to pick and run exactly one action toward the highest-priority goal.

With this setup, `PlanningAI` can flexibly switch between searching, attacking, defending, and more, all driven by simple flags, goal weights, and action costs.

5.4 GeneticPlanningAI

`GeneticAI` evolves the weights that drive a `PlanningAI`, letting bot teams "evolve" better behaviour over many simulated rounds.

5.4.1 Parameter Container

A small class, `GeneticAIParameters`, holds three things: a list of goal weights, a list of action weights, and an integer for how far to flank.

5.4.2 Wrapping PlanningAI

The main AI class, `GeneticPlanningAI`, extends `BaseAI` but actually just wraps a `PlanningAI` instance. In its constructor, it takes in a bot plus the three parameter values, stores them in Parameters, then calls `GenerateNewAI(bot)`, which builds `PlanningAI(bot, goalWeights, actionWeights, maxFlankPath)`. Every frame `UpdateAI()` hands control straight to that `PlanningAI`.

By shifting goal- and action-weight tuning into this automated loop, `GeneticAI` finds good parameter combinations on its own—no manual tweaking needed

5.5 Evolution Algorithm

The evolution system uses a basic genetic algorithm (GA) [9] to automatically tune each team's `PlanningAI` weights. The idea is to create a team where each bot has evolved into a "role". For example, each bot of a team can favour different goals such as flanking the enemy, hiding or rushing the enemy head-on. Therefore, `PlanningAI` weights are represented as genes in the evolution algorithm. The majority of the algorithm itself is inspired by the GeneticSharp's algorithm with minor tweaks [10]. The library offers great customisation when it comes to parts like fitness, crossover or selection; however, the library was insufficient when it comes to customising the actual algorithm. It was necessary to execute the crossover, mutation and selection, apply these chromosomes to a new generation and wait for the round to finish, since fitness was based on the performance of each team in the given round. Waiting for the processes to end and re-ordering these processes was not possible with the library.

5.5.1 Setup and Parameters

The `EvolutionAlgorithm MonoBehaviour` has two main parameters:

- `mutationRate` - This parameter determines the chance for a chromosome to change. The mutation rate has a 0.98 multiplier applied every round, so that as the algorithm progresses, the mutation is less likely since the later rounds should obtain high-quality chromosomes that shouldn't change too much.
- `crossoverRate` - This parameter represents the chance for two parent chromosomes to swap genes rather than copy directly

5.5.2 Chromosome Encoding

Each team is represented by a chromosome, which is just a flat array of floats (Genes) of length `teamSize × 8`. Every 8 genes correspond to one bot that has its AI initialised using `PlanningAI` with altered weights:

- Genes 0–4: These represent all five goal weights of `PlanningAI`. That is: stop, rush, flank, cover and wait for a teammate.
- Genes 5–6: These represent just two actions: help a teammate and retrieve a power-up. These have lower weight in the original `PlanningAI` than others, since it seemed beneficial for bots to aid their teammates and pick up power-ups.
- Gene 7: This gene represents the maximum distance for a flank to be admissible. If the route to the enemy using a particular flank found by the pathfinder is over this threshold, the flank is ignored.

5.5.3 Initial Population

On the very first simulation round, `StartEvolution(teamCounts)` builds a list of `populationSize` chromosomes, each filled with random genes valid genes within their valid range. In the entire experiment, there were 12 teams. The population was of size 11, and 1 team had AI based on default `PlanningAI`, which was used as a base for the fitness function. Unity fits only 12 teams since each team was fitted into its own layer, and their bullets had their own layer. Therefore, the number of teams is limited by the number of possible Unity collision layers.

5.5.4 Applying Parameters

Before each round, `ApplyChromosomesToTeams()` decodes every chromosome into a list of `GeneticAIParameter` (one per bot) using `GetBotParameters()`, then constructs a new `GeneticPlanningAI` for each bot with those weights. This ensures each team runs with its own unique set of genes each round of the evolution algorithm.

5.5.5 Scoring & Fitness

After a round ends, `RecordScores()` computes a normalised team score from three metrics: enemies killed, enemies found and surviving bots. Each is weighted kills by 0.5, finds by 0.1, survives by 0.5, and has a win multiplier of 1.2× if the

team won. boosts the raw sum before dividing by the maximum possible score. These normalised scores accumulate over a window of 6 rounds. Average score is calculated per round. The difference between the average score of `PlanningAI` team and the average score of a given `GeneticPlanningAI` team is the actual fitness. This is so that `GeneticPlanningAI` can be actually considered an evolution of `PlanningAI`.

5.5.6 Convergence

The genetic algorithm converged quite fast. It is probably due to its strong elitist selection, which rapidly purged any diversity. Moreover, fitness was defined solely as the margin by which a team outperformed the already well-tuned `PlanningAI`, leaving potentially little remaining performance head-room. Finally, brief evaluation windows (six rounds per generation) probably caused the algorithm to "lock in" on marginal gains within just ten generations (60 rounds).

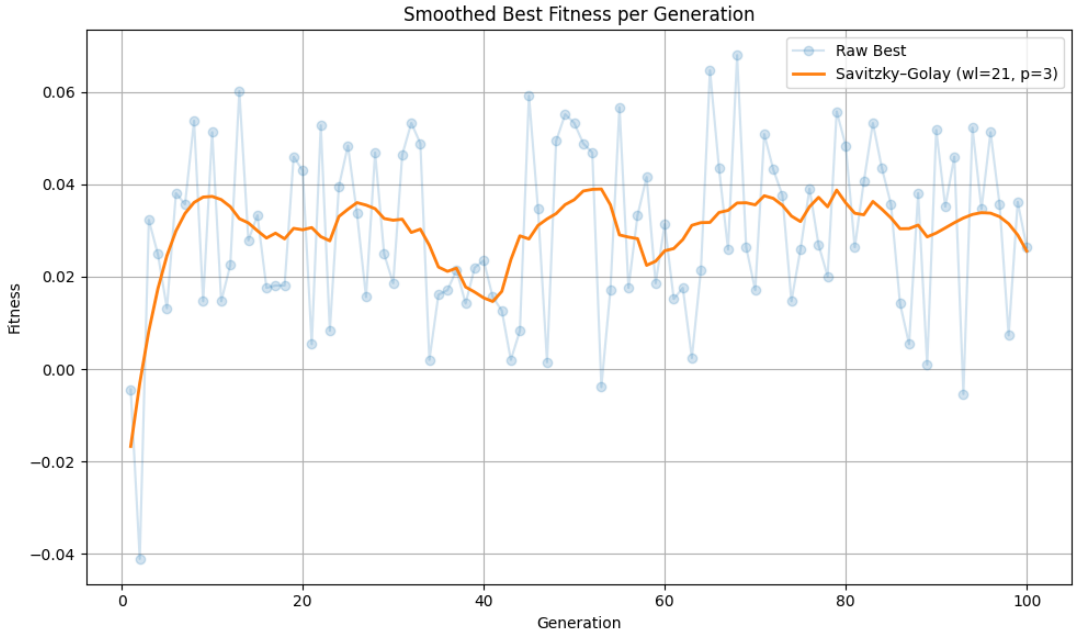


Figure 5.2 Convergence of the evolution.

5.5.7 Selection

Selection for the next generation is based on following principles:

- Elitism: The top two chromosomes copy directly into the next generation. Single best chromosome is mutated and it is also added to the next generation.
- Tournament selection: Picking parents by randomly sampling two chromosomes and choosing the fitter one, repeated to get two parents.
- With probability `crossoverRate`, parents exchange genes one by one; otherwise, each child is just a clone of a parent.

- Each child then calls `Mutate(mutationRate)`: with that probability, any gene can be reset to a new random valid value (same ranges as in initialisation).

5.5.8 Crossover

The crossover operator in our EvolutionAlgorithm implements a uniform crossover. When two parent chromosomes are selected, they undergo crossover with probability 0.8 a random check (50/50) decides whether the first child inherits from parent A and the second child inherits from parent B or vice versa, producing two offspring whose gene arrays are composed of a mix of the parents' genes on a per-gene basis [9]

5.5.9 Next Generation

Once the new population fills up, it replaces the old one. The evolution algorithm then continues over successive generations until the simulation ends by looping through rounds that are set in the Unity inspector. Scoring team performance, and using selection, crossover, and mutation crafts better parameter sets. The evolution algorithm drives teams of bots to "learn" effective goal and action weightings.

5.6 NeuralAI

The `NeuralAI` class implements a neural-network-driven agent for decision making in a team-based game environment. It uses a two-layer feed-forward neural network [11] as its policy function, mapping the agent's observed state to a probability distribution over possible actions. In other words, the network maps the bot's current state to action probabilities using a softmax output layer

5.6.1 Input Layer and State Encoding

The `NeuralAI's` input layer takes a vector of numerical features representing the bot's state and situational awareness. In the code, the input vector length is set to 15, composed of several parts capturing the bot's environment and status. Key components include:

- Bot State Flags (6 values): Six binary features indicate important boolean states of the bot. The flags include `FoundEnemy`, `TeammateFoundEnemy`, `TeammateAidArrived`, `TeammateDied`, `FoundPowerUp`, `RetrievedPowerUp`. These flags are encoded as 1.0 or 0.0 in the first six input slots.

- Distances to Teammates (up to 3 values): The next part of the input encodes the Euclidean distances from this bot to each of its teammates. For a team size of 4 (this bot plus three others), there are three distance values. These distances are computed from the bot's current position to each teammate's position.

- Bot's Own Position (2 values): The bot's current world coordinates `(x, y)` are included next. These continuous values situate the bot within the game map.
- Enemy Position (2 values): If the bot or a teammate has spotted an enemy, the estimated enemy coordinates (x, y) are included. These coordinates inform the network where a potential threat is.
- Destination Position (2 values): The final two inputs represent the bot's current navigation target or destination in world coordinates. The simulation sets a destination (e.g. a point to move toward, such as a power-up location or a search area in the maze) for the bot. If there is an active destination (e.g. provided by pathfinding or search routines), its coordinates are encoded; if not, this may be `(0,0)`. This guides the network on whether the bot is currently moving somewhere meaningful.

All these parts are concatenated into one input vector (length 15). Thus, at each decision step, the `GetStateInputs()` method of `NeuralAI` assembles this feature vector reflecting the bot's immediate situation. This rich state representation allows the neural network to consider not only the bot's own status but also team dynamics and goal locations when choosing an action.

5.6.2 Hidden Layer

The input layer connects to a hidden layer of neurons (32 by default). Each hidden neuron computes a weighted sum of all input features plus a bias term. The `NeuralAI` uses the hyperbolic tangent (tanh) as the activation function for hidden neurons. Tanh squashes the summed input into a range (-1 to 1), introducing nonlinearity. The hidden layer is fully connected to the output layer, which in this implementation consists of 4 output neurons (since the bot has 4 distinct high-level actions to choose from). Each output neuron j computes a weighted sum of all hidden activations plus its bias. [11] These raw sums (often called logits) are then fed into a softmax function to produce a normalised probability distribution over actions. Softmax ensures all output probabilities are in [0,1] and sum to 1.

5.6.3 Forward Propagation and Decision Making

When it's time for the bot to act, the `NeuralAI` performs a forward pass through the network. It calls a function that takes the current state input vector and computes the hidden activations and then the output probabilities as described above.

5.7 NeuralAI training

5.7.1 Reinforcement Learning Setup in the Simulation

Training the `NeuralAI's` neural network relies on a policy gradient reinforcement learning approach, specifically similar to the REINFORCE algorithm [11].

The key idea is to update the network's weights in a direction which promotes chosen actions with higher observed rewards. In this implementation, training is done episodically and centrally: after a set of game rounds (episodes), a single "master" network is updated with the cumulative experience, and then those updated weights are distributed to all the team's agents. Reward Calculation and Normalisation: Unlike a simple game where win/lose might be a clear reward, this environment provides a richer set of performance metrics for each team. After each round (episode), the simulation computes a score for each team based on that round's outcomes. In the provided code, the team's score is a weighted sum of: enemies killed, enemies spotted, and bots survived. Specifically, each enemy kill contributes 0.5 to the score, each enemy found (spotted) contributes 0.1, and each surviving team member contributes 0.5. These weights reflect the priorities of the task (eliminating enemies and keeping allies alive are high-value, hence 0.5 each, whereas simply spotting enemies is useful but lower impact). The sum is a raw score for the round. This raw score is then divided by the maximum raw score among all teams in that round to yield a normalised performance score between 0 and 1. Normalisation ensures that the best-performing team of a round gets a score of 1.0, and others get a fractional score representing their performance relative to the best. This is important because it scales the reward to a consistent range regardless of how high or low the absolute numbers of kills or deaths might go. The training is set to occur not every single round, but every 3 rounds (episodes). The system accumulates the normalised scores of the last three rounds for each team, then computes an average score for each team over this mini-window. Using an average smooths out round-by-round variability and noise. It is a form of reward smoothing.

5.7.2 `TrainNeuralNetwork` and `TrainOnReward` functions

The `NeuralAI` training uses a policy gradient approach [12], specifically the REINFORCE algorith [11]. In the code, `TrainNeuralNetwork()` (in Simulation) calls `ai.TrainOnReward(reward)` on the `NeuralAI` instance (where reward is the team reward described above). This triggers the policy gradient weight update using all the experiences accumulated in the `NeuralAI` since the last update.

Using Accumulated Experiences: The `NeuralAI` has stored each state it encountered and the action it took (the "trajectory" of the last few rounds). The training iterates over each stored experience. For each experience, it does the following:

- Forward Pass:

  It feeds the state into the current neural network to get the output probabilities. This step is necessary to evaluate the policy's current behavior on past states.[12]

- Calculate Policy Gradient Error [11]:

  The algorithm then compares the chosen action in that experience to the probability distribution. For the experience's action, the ideal policy would have probability 1 and 0 for others (in hindsight for learning). The difference between the one-hot ideal and the current probabilities gives an error signal.

This vector is then scaled by the reward. This constitutes the gradient of the log-probability weighted by the reward, since $\nabla_\theta \log P_\theta(\text{action} \mid \text{state}) \cdot r = \Big(\mathbf{1}_{[\text{action}]} - P_\theta(\cdot \mid \text{state})\Big) \cdot r$ for non-taken actions. This approach follows the standard policy gradient update rule: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_e \mid s_e) \cdot r$, which moves the policy parameters in the direction that would make the chosen action more likely if reward was positive (or less likely if it was negative). Intuitively, if the reward is positive, this will increase the weights' support for the taken action and decrease it for the others proportional to their probabilities; if the reward is negative and the team did poorly, it will do the opposite, discouraging the actions that were taken.

- Backpropagate and Update Weights [11]:

  Using the error signal computed above (often called the policy gradient), the `NeuralAI` then updates its neural network parameters via backpropagation. First, it updates the second layer (hidden-to-output weights). For each output neuron, the gradient for that neuron's bias is just the error for that action. Implemented as `weights2[i,j] += learningRate * dOutput[j] * hiddenAct[i]` for each hidden i and output j, and similarly adjust each output bias by `+ learningRate * dOutput[j]`. This is a gradient ascent step on the log-likelihood objective (since they add the gradient; equivalently they could subtract the negative gradient).

- Propagate to First Layer [11]:

  After updating the output layer, the algorithm computes the gradient for the hidden layer. This involves propagating the output error back through the weights to get an error term for each hidden neuron. The code computes this as `dHidden[i] = (1 - hiddenAct[i]*hiddenAct[i]) * sum` (sum is over j of `weights2[i,j] * dOutput[j]`). Now `dHidden[i]` represents the influence on the hidden neuron's output on the final reward-weighted error.

  Using these hidden-layer gradients, the first layer weights are updated: for each input feature k and hidden neuron i, like this:

  `weights1[k,i] += learningRate * dHidden[i] * state[k]`.

  Similarly, each hidden bias gets `+= learningRate * dHidden[i]`. In effect, if a particular input feature was present (for example, found enemy flag = 1) and that led to a successful action, the weights connecting that feature to the hidden units that eventually influenced the chosen action will be nudged up, and vice versa for negative outcomes.

After the training update, the experience memory in each `NeuralAI` is cleared (`ClearExperiences()`), so that the next set of rounds will collect fresh experiences under the new policy. Clearing old experiences is important to avoid applying stale data from a previous policy to updates for a new policy (which could introduce bias or divergence). This process is repeated for every recorded experience. Because the same reward is used for all time steps of the episode(s) in the update (in this implementation), it assumes an episodic reward scenario where all actions in those rounds are credited or blamed equally for the final outcome. This is a simplification. Conceptually, this training method is making the policy more

likely to repeat actions that led to wins and less likely to repeat those that led to losses. Over many rounds, the `NeuralAI` should increase the probability of effective strategies (for example, perhaps flanking gives higher survival and kill rates, so the network will start favouring the "flank" action in relevant states if that correlation is learned).

5.7.3 Training Cycle and Weight Updates Frequency

The simulation integrates this training step into its main loop. After each training round (every third round), the updated neural network weights (now the master network) are saved and then applied to all bots of Team 0. Specifically, after training, the code clones the learned `NeuralAI's` weights into a stored `NeuralNetworkData` object `masterNet`. `ApplyNeuralNetworksToTeam()` is called after, which re-initializes each bot's `NeuralAI` with the updated weights. This ensures that all bots start the next round with the improved policy. In essence, Team 0's bots share a common policy that is being learned and broadcast to them after each update.

Additionally, the code calls `DumpMasterNetwork(...)` to save the current network parameters to a file for offline analysis or future use, and also logs cumulative win statistics. These measures allow monitoring of learning progress over time (for example, one could inspect the JSON weight files to see how weights change, or check the win statistics to see if Team 0's win rate is improving).r each team over this mini-window. Using an average smooths out round-by-round variability and noise – it's a form of reward smoothing

6 Game flow

6.1 Game Manager

The `GameManager` controls the main flow of the game. It takes care of starting matches, setting up teams, preparing the maze, and ending the round when needed. While bots and teams focus on their own tasks, the `GameManager` watches over the entire match and decides when things should begin or stop. It does not control bots directly but makes sure everything is ready for them to work properly.

6.1.1 Match Flow and Game State

The game runs in steps. First, it starts the setup. Then it begins the battle. Finally, it checks if the match should end. The `GameManager` keeps track of this using flags like `gameEnded`, `battleStarted`, and simulation. These help it know what should happen in each frame. For example, bots only start moving after `battleStarted` is true. When `gameEnded` becomes true, the AI and `pathfinding` are skipped to save time and performance.

6.1.2 Working with Maze and Teams

One of `GameManager's` jobs is to connect different systems. For example, it asks the Maze to create spawn points and open up editing. After that, it calls `MazeEditEnded()` to lock in the setup and create all the teams. Once teams are created, the `GameManager` gives each team a copy of the maze. This is important because bots might change the maze temporarily when doing flanks.

6.1.3 Deciding When to End the Match

Every frame, the `GameManager` runs `CheckGameEnded()`. This function checks a few things:

- Has the time limit passed?
- Are all bots idle (not moving or fighting)?
- Is there only one team left?

If any of these are true, the round ends. This lets matches stop automatically when they are truly finished, not just based on a timer.

6.1.4 Preparing Data for Pathfinding

The `GameManager` doesn't do pathfinding itself. Before running AI updates, it calls `UpdatePathfinderPositions()` so every team knows where their bots are and their targets. After this setup, the `GameManager` uses `Parallel.ForEach()` to run pathfinding and AI updates at the same time for all teams. This saves a lot of time, especially in matches with many bots.

6.1.5 Ending the Game and Cleanup

When the match ends, the `GameManager` destroys all bots. This clears up memory and avoids leftover data in the next round. It also resets win tracking and other match stats.

6.2 Simulation

The Simulation system automates running game rounds so we can measure and improve our AI without manual play. It's configured via public fields in Simulation.c, including:

- `simulationRounds` (how many rounds to run),
- `simulationDelay` (pause between rounds so everything is initialised properly),
- `simulationSpeed` (time-scale multiplier),
- EvolutionAlgorithm and NeuralTraining (toggles for GA or NN learning)

When `StartSimulation()` is called, it puts the game into AI-only mode in `GameManager` and fires off the `RunSimulation()` coroutine. Each loop iteration represents one round. That includes starting the game, which sets up teams, bots, FOVs and a few other variables. After each round finishes, Simulation collects each team's performance metrics. Specifically, it records the number of kills, number of surviving bots, and the ID of the winning team. Collected metrics are stored as json for each round for each team so that the metrics can be processed and plotted in python.

7 Experiments

To evaluate the performance of different AI strategies implemented in the game, an automated tournament system was designed. The goal of these experiments was to assess the strengths and weaknesses of each AI across multiple randomised environments and team configurations, focusing on a few simple metrics.

7.1 Experimental Setup

The tournament setup of the game was extended with a simulation controller. For each round, a new randomised maze was generated using Kruskal's algorithm with additional loop carving. Six teams were instantiated at random spawn points, each with 4 bots. Each team was randomly assigned one of the available AI types. The game logic automatically tracked:

- Kills: number of enemy eliminations by each team,
- Survivors: bots that remained alive at the end of the round,
- Winner: the team that outlasted all others,
- AI type: explicitly recorded at the start of each round, since bots may be destroyed before data collection.

Each round's results were serialised as JSON and stored to disk. The Python script tournament.py parsed these results and computed various statistics and plots.

7.2 Metrics

Following performance metrics were used:

- Win rate: proportion of rounds won by a given AI type,
- Mean kills count: indicators of combat and evasion effectiveness,
- Rolling win rate: smoothed estimate of recent performance across rounds,

These metrics were chosen to balance short-term tactical success (kills) and long-term strategic survival (wins).

7.3 Results Summary

In the tournament with a random choice of team AI, the results were following:

- RandomAI:

  Significantly behind all others. The lack of strategy led to erratic and ineffective behaviour.

- NeuralAI:

  Underperformed due to abrupt changes in its behaviour during the tournament. Changes in actions too abruptly lead to chaotic behaviour. On top of that it seems like it was lacking variance in its action decisions as there was too strong a bias towards flanking, which might indicate that the training was poor.

- AutomataAI:

  Performed moderately well, especially in predictable scenarios. Lacked flexibility in complex situations. It is probably the most useful out of the bunch. Being so easy to create and so predictable, and yet its performance is quite dependent, makes it probably the best value when it comes to programming effort and the quality of the AI

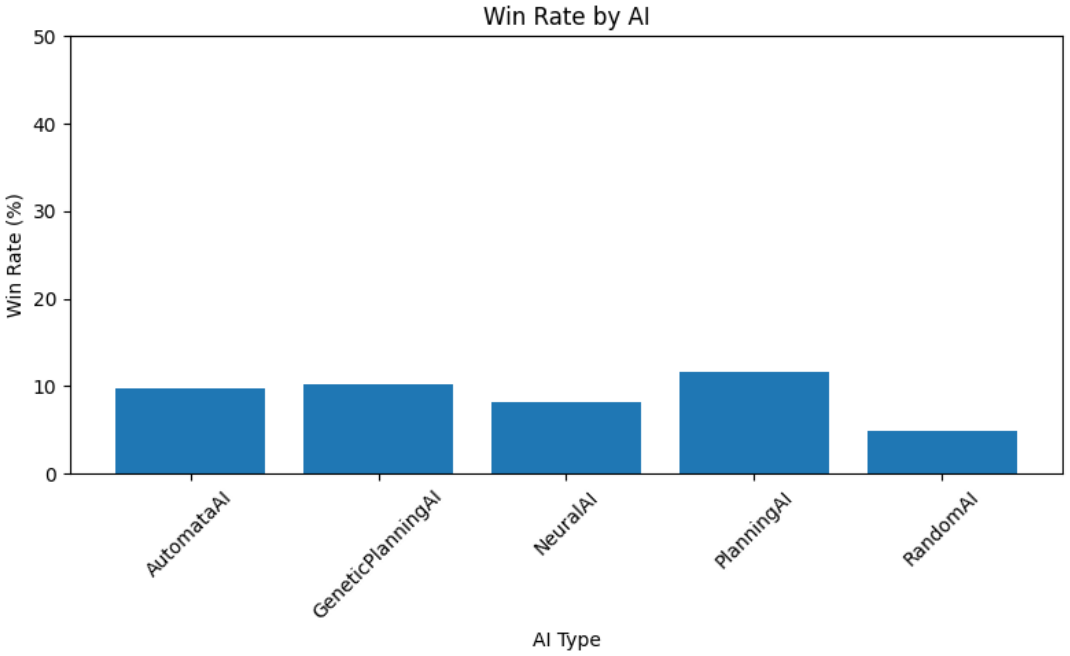
- GeneticAI:

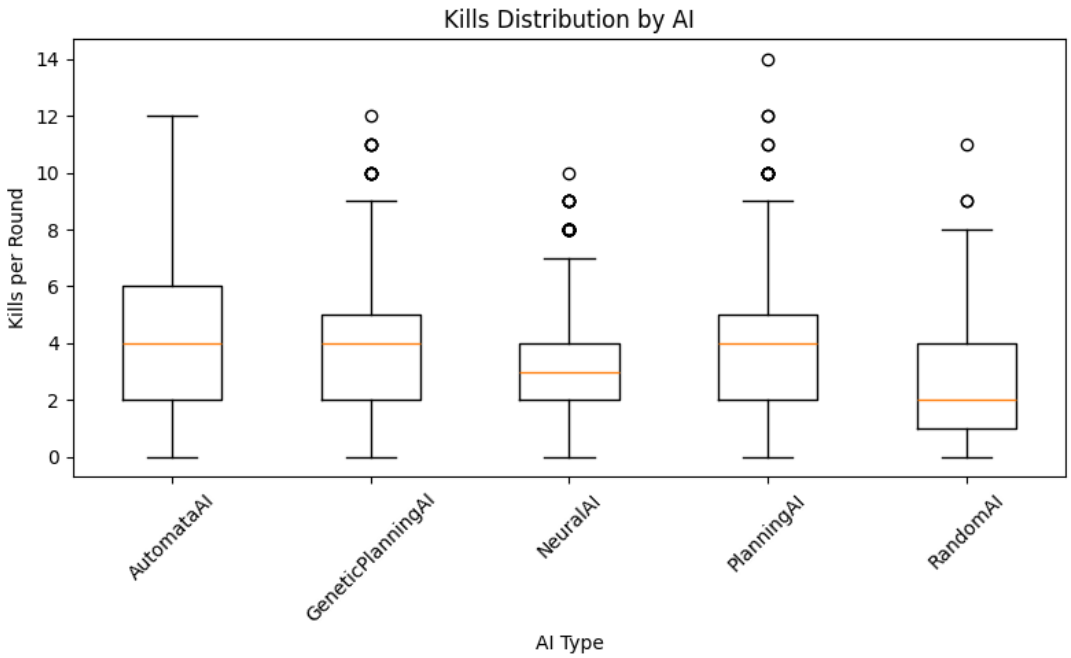
  This team showed similar capabilities like `PlanningAI`, however it seems like it was not as well suited to tournaments with 6 various teams and seems a little worse off than `PlanningAI`. There was probably too much emphasis on beating just `PlanningAI` in the fitness function.

- PlanningAI:

  This AI consistently outmaneuvered opponents by leveraging symbolic reasoning and team-aware planning

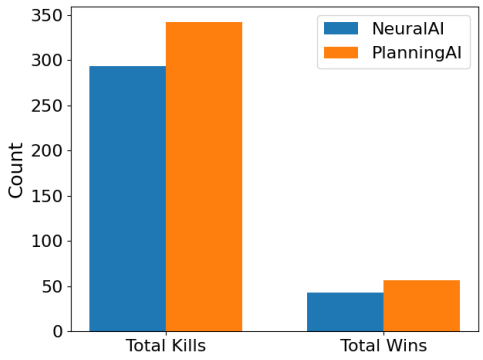
- Results:



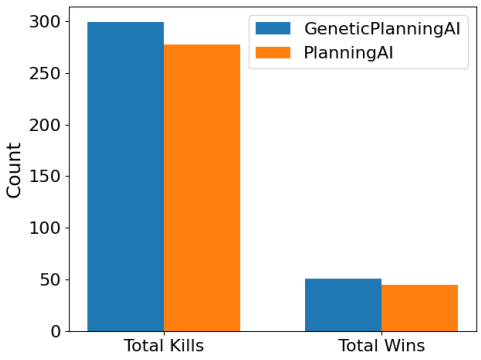


7.4 1v1 results

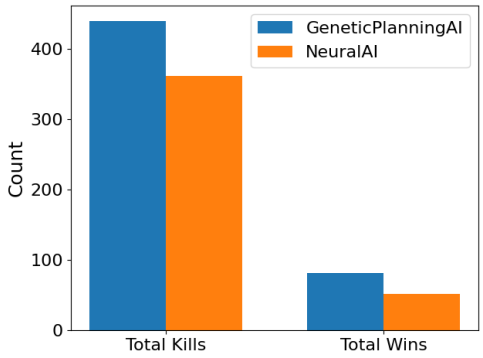
There was another experiment that was structured a bit differently. This time, only 100 rounds were conducted, where only two teams of four bots competed against each other. The fact that `PlanningAI` dominated the tournament with a random pick of AI was suprising. That is the reason for this tournament and its structure. Interestingly although in mixed-AI tournaments `PlanningAI` appeared stronger, in direct combat, `GeneticPlanningAI` defeated it. This shows that while `PlanningAI` is more stable across varied opponents, the evolved `GeneticAI` version develops strategies better tuned specifically for fighting `PlanningAI` itself.



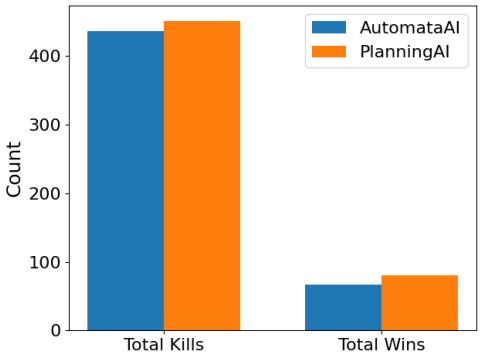
(a) NeuralAI vs PlanningAI



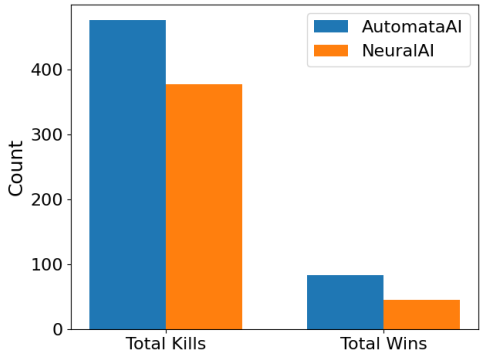
(b) GeneticPlanningAI vs PlanningAI



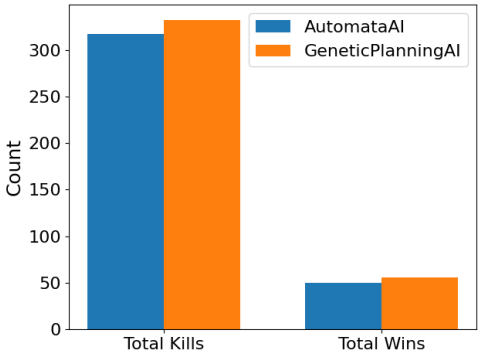
(c) GeneticPlanningAI vs NeuralAI



(d) AutomataAI vs PlanningAI



(e) AutomataAI vs NeuralAI



(f) AutomataAI vs GeneticPlanningAI

Figure 7.1 Results of 1v1 experiment

8 Optimizations

To support fast-paced, team-based AI combat in a maze environment, targeted performance optimizations were implemented in the simulation framework. These optimizations affect mostly the bot AI evaluation and pathfinding computations.

8.1 Parallel Execution

Performance boost comes from executing each team's pathfinding and AI updates in parallel. In the `GameManager` class, the main simulation loop utilises this construct: `Parallel.ForEach(Teams, team => team.UpdatePathfinding(); team.UpdateAI(); );`

This reduces latency by distributing expensive updates—such as A* pathfinding and especially the `UpdateAI` function. These updates are done in parallel for each team, as each team has its own copy of a pathfinder, since flanking can actually alter the maze for the pathfinder momentarily for flanking. Since non of the Unity API can be accessed out of the main thread it is necessary to load all the necessary data to Pathfinder beforehand. Positions of bots and enemies, their destinations and their "need pathfinding" flags are cleared and loaded before each pathfinding update. On top of that, bots indicate when they need new paths using a flag `NeedPathfinding`. This avoids recalculating paths for idle or stationary bots.

8.2 Conditional AI Updates

In the Bot.cs class, the `UpdateAI()` method does not blindly call the AI logic every frame. Instead, it includes a lightweight check to ensure updates are triggered only when meaningful state transitions occur. There are following checks: `CheckEnemyKilled()`, `CheckNavigation()` and `CheckTeam()`. Killing an enemy, navigation changes like reaching your destination, team changes like teammate dying, finding an enemy or arriving for help, and finding or retrieving a power up are all changes that trigger an AI update. This logic prevents redundant computation and preserves CPU cycles by skipping updates when the bot's state has not changed in ways that would influence AI output.

Conclusion

This thesis described the design and implementation of a 2D top-down shooter game in Unity, built with the goal of testing and comparing multiple AI approaches in a controlled, maze-based environment. The game serves both as a functional prototype for a game and as an experimental platform for AI behavior in scenarios involving movement, partial information, and team-based decision-making.

The implementation included several distinct AI systems: a Random AI used as a baseline, an Automata-based finite-state machine, a symbolic planner (PlanningAI) based on GOAP, a GeneticPlanningAI which evolved goal and action parameters over generations, and a NeuralAI trained through policy gradient reinforcement learning. Each of these was integrated into a modular architecture that cleanly separates decision logic, bot behavior, movement, and simulation control. To support large-scale testing, the system was extended with a simulation mode that runs repeated matches with randomized mazes. This setup enabled fair comparisons between AIs by averaging performance over many rounds. Performance optimizations such as multithreaded pathfinding and conditional AI updates were introduced to make large simulations feasible. Experimental results showed clear differences between the methods. RandomAI, as expected, performed poorly. AutomataAI provided decent results with minimal complexity. PlanningAI showed the most consistent and effective behavior in team settings. GeneticAI performed similarly to PlanningAI in some cases but showed limitations in more diverse tournaments. NeuralAI lagged behind overall, and its performance appeared unstable, likely due to insufficient exploration and high variance during training.

In summary, the project demonstrated a working testbed for comparing AI strategies in a tactical game setting. While not every AI approach performed equally well, the implementation provided useful insights into their strengths, weaknesses, and trade-offs in a shared environment. Future work could focus on improving the training stability of NeuralAI, exploring more advanced genetic operators for GeneticAI, or incorporating additional mechanics into the game to push the limits of AI adaptability.

Bibliography

1. Unity Technologies. *Unity Real-Time Development Platform* [https://unity.com]. [N.d.]. Version used: 202X, Accessed: 2025-07-13.
2. Hart, Peter E.; Nilsson, Nils J.; Raphael, Bertram. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*. 1968, vol. 4, no. 2, pp. 100–107. Available from doi: 10.1109/TSSC.1968.300136.
3. Microsoft Corporation. *C# Language Specification* [https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/]. [N.d.]. Accessed: 2025-07-13.
4. Unity User: Rhonabwy. *Unity 2D: Visualise a field of view cone* [https://discussions.unity.com/t/unity-2d-visualise-a-field-of-view-cone/236443/1]. [N.d.]. Unity Discussions, Accessed: 2025-07-13.
5. Kruskal, J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In: *Proceedings of the American Mathematical Society, 7*. 1956.
6. Booth, Taylor L. *Sequential Machines and Automata Theory*. John Wiley & Sons, 1967.
7. Orkin, Jeff. Agent Architecture Considerations for Real-Time Planning in Games. In: Young, R. Michael; Laird, John E. (eds.). *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*. AAAI Press, 2005, pp. 105–110.
8. caesuric. *mountain-goap* [https://github.com/caesuric/mountain-goap]. [N.d.]. Accessed: 2025-07-13.
9. Goldberg, David E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
10. Giacomelli, Henrique. *GeneticSharp: A Genetic Algorithm Library for .NET* [https://github.com/giacomelli/GeneticSharp]. [N.d.]. Accessed: 2025-07-13.
11. GeeksforGeeks. *Reinforce Algorithm* [https://www.geeksforgeeks.org/machine-learning/reinforce-algorithm/]. [N.d.]. Accessed: 2025-07-13.
12. Sutton, Richard S.; McAllester, David A.; Singh, Satinder; Mansour, Yishay. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: Solla, Sara A.; Leen, Todd K.; Müller, Klaus-Robert (eds.). *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*. The MIT Press, 1999, pp. 1057–1063. Available also from: http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

A User Documentation

The user interface (UI) provides a simple but effective way for users to configure the game before it starts. Having an accessible UI makes it easier to test different settings and evaluate bot performance under various conditions.

A.1 Simulation Startup

To run a simulation, the full Unity project (including all required libraries) must be opened using Unity version 2020.3.48f1. In the Unity Editor, select the "game" object in the hierarchy, then enable the "Simulation" checkbox in the Inspector. It can be found by clicking on Canvas in the hierarchy and in the inspector there is a script called `PlayButton` which holds the checkbox. Depending on the desired experiment, also check either "Evolution Algorithm" or "Neural Training" to activate the corresponding AI system. These can be found by clicking on Game in the hierarchy window and then navigating to the `Simulation` script in the inspector. In the same inspector there are also editable parameters such as simulation speed, number of rounds and the duration of a single round since there is a possibility that bots might take long time to find each other if only 2 last ones are left. Dumps of the simulation for neural training, evolution or the tournament reside in Assets folder in the project. Dumps can be processed with the python scripts.

A.2 Game Startup

To run the game, navigate to the AIBattleRoyale folder and launch the executable named `RProject.exe`. No installation is needed—just double-click the file and the game will start. Make sure your system meets Unity's runtime requirements and that you allow any permissions if prompted.

A.3 Menu Structure

The main menu consists of a title screen, where clicking the left mouse button anywhere on the screen makes the configuration UI visible. This part of the interface includes dropdowns, sliders, and labels which allow the user to customise each team and the environment.

A.3.1 Team Configuration

For each team (up to six total), the user can set two parameters:

- Squad Size – A dropdown with values from 0 to 8 that determines how many bots the team will have. Setting this to 0 disables the team. Note that dropdowns should be filled in order of team number, otherwise the game will move the counts automatically.

- AI Type – A dropdown for selecting the control logic for the team's bots. If nothing is selected the default is AutomataAI. Options include:

  Random AI Neural AI Automata AI Planning AI Genetic Planning AI

A.3.2 Maze Configuration

A single slider is used to select the maze size. The value ranges from 10 to 50. Next to the slider is a dynamic label which updates to show the current size.

A.3.3 AI Toggle

The checkbox in the upper right corner simply determines whether the user is taking part in that round. If the checkbox is ticked, only AI bots will play in that round; if it is unticked, the user can control one of the bots and have AI teammates.

A.3.4 Play Button

Clicking the play button in the lower right corner generates the maze is generated along with the spawn points that are represented by colored balls.

A.4 Maze Editing

Before the game begins, the user has the option to manually edit the maze. This functionality is handled by the `MazeGenerator` class, which includes a special edit mode that is active during the configuration phase of the game.

A.4.1 Perfect Maze Toggle

This checkbox uses an alternative algorithm that creates a maze where paths don't create cycles

Figure A.1 Setup

A.4.2 Adding and Removing Walls

The user can click directly on the maze grid to modify walls:

Left Mouse Button (LMB): Adds a wall to the cell the mouse is currently hovering over. This action only works if the targeted cell is a valid "Path" cell and not on the outer boundary.

Right Mouse Button (RMB): Deletes a wall from the targeted cell, converting it into a traversable "Path" cell.

Be careful not to box in a team or create dead zones in the maze.

A.4.3 Moving Team Spawn Points

The maze also allows manual placement of spawn points for each team. While in edit mode, the user can press one of the number keys (1 through 6) to select a team. Then, by clicking on a valid cell, the spawn point for that team is moved to the clicked location.

Each team has a corresponding Ball object that visually marks its spawn position in the maze. These objects are updated in real time as the user clicks to reposition them.

A.4.4 Finalizing the Maze

Once the user is satisfied with their modifications, pressing the Return key exits edit mode.

A.5 Gameplay Mechanics

When you take control of one bot, your interaction is limited to movement, aiming, and shooting.

A.5.1 Movement

You move by holding W/A/S/D, which directly applies a physics force each frame:

```
Body.AddForce(new Vector2(x, y));
```

The variable d combines your bot's speed setting and `Time.deltaTime`, so motion feels smooth and consistent. There's no built-in sprint or crouch—just constant, physics-driven acceleration.

A.5.2 Aiming

Your bot always faces either where you clicked or, if you haven't clicked yet, your most recent movement direction. Internally, the code in `LateUpdate()` reads your mouse position in world space, converts it into an angle, and then sets the bot's `transform.rotation` accordingly. This rotation not only turns the sprite but also orients the firing point for bullets.

A.5.3 Shooting

Click the left mouse button to fire. When that runs, the Shooting script spawns a projectile prefab in front of your bot. There's no reloading or ammo count—just as many clicks as you like.